DISCRETE EVENT SIMULATION USING REPAST JAVA: A DISCRETEEVENTSIM TUTORIAL

TIM SWEDA

0. Before Getting Started

This tutorial assumes that Repast Simphony (RepastS) 2.0 is already installed and includes the DiscreteEventSim model. If not, the latest version of RepastS can be downloaded from the Repast website, and DiscreteEventSim is available here.

1. Introduction to Discrete Event Simulation

Most agent-based models are examples of discrete *time* simulations. In them, time is discretized into uniform intervals (called "ticks" in Repast), and a set of actions takes place at the beginning of each interval. After a set of actions is completed, the simulation advances to the beginning of the next time interval and carries out a new set of actions.

By contrast, discrete *event* simulation proceeds according to a chronological sequence of events rather than time steps. When a new event occurs, the simulation clock is updated based on the time of the event and a set of actions is performed. Such a framework is more suitable for modeling systems in which events can occur in continuous time since there are no restrictions on when events may occur.

2. The DiscreteEventSim Model

RepastS provides a nice platform for discrete event simulation, and the DiscreteEventSim model from the models library offers a convenient set of tools for building discrete event systems without having to start from scratch. In the remainder of this section, we will explore what is included in DiscreteEventSim and discuss how to implement systems that are more complicated than the $M/M/1$ queue shown in the model.

2.1. **Entity class.** In a discrete event system, an *entity* is any object that moves through the system. For example, if the system is a checkout lane at a supermarket, the entities could be the customers. They enter the system when they bring their groceries, and they leave the system after paying and getting their groceries bagged.

The `Entity` class in DiscreteEventSim has just two variables: `priority` and `timestamp`. The value of `priority` determines how an Entity is treated if the system prioritizes some Entities over others, but otherwise it is not used. It has corresponding `set` and `get` methods and can optionally be initialized in the Entity's constructor method. The `timestamp` variable is useful for tracking how long the Entity spends in part of the system, and its value can be set or obtained with the `stamp` and `read` methods, respectively.

```
1  public class Entity {
2
3     private double priority; /* Priority of entity (important if using
4        priority queues) */
5     private double timestamp; /* Useful for calculating duration of an
6        activity (such as waiting in a queue) */
7
8     public Entity() {
9        priority = 0;
10    }
11
12    public Entity(double pr) {
13       setPriority(pr);
14    }
15
16    public void setPriority(double pr) {
17       priority = pr;
18    }
19
20    public double getPriority() {
21       return priority;
22    }
23
24    public void stamp(double time) {
25       timestamp = time;
26    }
27
28    public double read() {
29       return timestamp;
30    }
31
32 }
```

2.2. **Stat class.** Collecting statistics is important when analyzing a simulated system, and the `Stat` class provides an easy way of doing so. Each Stat keeps track of a single system statistic and is tabulated differently depending on whether the statistic is continuous-time or discrete-time. Continuous-time statistics, whose values change continuously (such as the average number of Entities in the system), are recorded using the `recordCT` method. Discrete-time statistics, whose values change at discrete intervals (such as the average time an Entity spends in the system), are recorded with the `recordDT` method.

```
1  public class Stat {
2
3     private double total, totalSq, count, lastTime;
4
5     public Stat() {
6        initialize();
7     }
8
9     public void initialize() {
10       total = 0;
11       totalSq = 0;
12       count = 0;
13       lastTime = DESimBuilder.schedule.getTickCount();
14    }
15
16    // Record continuous-time statistic
17    public void recordCT(double value) {
18       double currentTime = DESimBuilder.schedule.getTickCount();
19       total += value*(currentTime-lastTime);
20       count += currentTime-lastTime;
21       lastTime = currentTime;
22    }
23
24    // Record discrete-time statistic
25    public void recordDT(double value) {
26       total += value;
27       totalSq += value*value;
28       count++;
29    }
```

The `getAverage` method returns the average value of the statistic tabulated by the Stat, and for discrete-time statistics, the `getStDev` method returns the standard deviation.

```
1     // Return average
2     public double getAverage() {
3        if (count > 0)
4           return total/count;
5        else
6           return 0;
7     }
8
9     // Return standard deviation (relevant only for discrete time statistics)
10    public double getStDev() {
11       if (count > 0)
12          return Math.sqrt(totalSq/count);
13       else
14          return 0;
15    }
16
17 }
```

2.3. **Queue class.** A *queue* is an object that holds entities as they wait in the system. Going back to the supermarket example referenced earlier, the line at the checkout counter is a queue. When a new customer arrives, he or she can checkout immediately if there is nobody else in line, but must otherwise wait for other customers who arrived earlier to checkout first.

The `Queue` class has many different components, so we will look at it in parts rather than all at once. We begin by looking at the three class variables: `type`, `length`, and `waitTime`. A Queue's `type` determines how waiting Entities are ordered. It can be "FIFO" (first-in-first-out), "LIFO" (last-in-first-out), or "Priority" (Entities with higher `priority` are serviced first). (Other options are possible but are not currently implemented.) The `length` and `waitTime` are Stats that keep track of the length of the Queue and the time Entities spend waiting in the Queue, respectively.

```
1  public class Queue extends ArrayList<Entity> {
2
3     private String type; /* Type of queue:  FIFO (default), LIFO,
4      or Priority */
5     private Stat length; // Number of entities waiting in queue
6     private Stat waitTime; // Waiting time of entities in queue
```

There are two constructor methods, one for when the Queue's `type` is initialized and another for when it is not. If no `type` is specified or it is not one of the three recognized

Strings, then the Queue defaults to "FIFO." The last five lines of each constructor initialize the two Stats and add the Queue (along with both Stats) to the system.

```
1     public Queue() {
2        type = "FIFO";
3        length = new Stat();
4        waitTime = new Stat();
5        DESimBuilder.qList.add(this);
6        DESimBuilder.sList.add(length);
7        DESimBuilder.sList.add(waitTime);
8     }
9
10    public Queue(String type) {
11       this.type = type;
12       // If type is not recognized, default to FIFO
13       if (type != "FIFO" && type != "LIFO" && type != "Priority")
14          this.type = "FIFO";
15       length = new Stat();
16       waitTime = new Stat();
17       DESimBuilder.qList.add(this);
18       DESimBuilder.sList.add(length);
19       DESimBuilder.sList.add(waitTime);
20    }
```

Entities are removed from and added to the Queue using the `pop` and `enqueue` methods, respectively. These methods take into account the Queue's `type` and handle Entities accordingly. The two Stats are also updated automatically as Entities arrive and depart. The `skip` method lets the Queue know that an Entity did not have to wait so that it can update its `waitTime` Stat. It is equivalent to executing the `enqueue` and `pop` methods sequentially.

```
1     // Remove next entity from queue
2     public Entity pop() {
3        length.recordCT(this.size());
4        if (this.isEmpty())
5           return null;
6        waitTime.recordDT(DESimBuilder.schedule.getTickCount()-
7          this.get(0).read());
8        return this.remove(0);
9     }
10
11    // Add incoming entity to queue
12    public boolean enqueue(Entity e) {
13       length.recordCT(this.size());
14       double time = DESimBuilder.schedule.getTickCount();
15       if (type == "FIFO") {
16          e.stamp(time);
17          if (this.add(e))
18             return true;
19       }
20       else if (type == "LIFO") {
21          this.add(0, e);
22          e.stamp(time);
23          return true;
24       }
25       else if (type == "Priority") {
26          int i = 0;
27          while (e.getPriority() <= this.get(i).getPriority())
28             i++;
29          this.add(i, e);
30          e.stamp(time);
31          return true;
32       }
33       System.err.println("Unable to add entity to queue");
34       return false;
35    }
36
37    // Let queue know that an entity did not need to wait for service
38    //   (same as enqueue+pop)
39    public void skip() {
40       waitTime.recordDT(0);
41    }
42
43 }
```

2.4. **Resource class.** A *resource* is an object used by entities in a system. In the supermarket example, the checkout counter (or cashier) is the resource. Customers wait in line to use the checkout counter (i.e., pay for their groceries), and then they leave.

The `Resource` class has three variables: `numServers`, `numBusy`, and `utilization`. The value for `numServers` represents the total capacity of the Resource, and `numBusy` indicates the capacity currently in use and unavailable to arriving Entities. The Stat `utilization` keeps track of how much of the Resource's capacity is used by Entities. The constructor method sets the capacity of the Resource, initializes it as unused (i.e., all of its capacity is available), and adds it along with the `utilization` Stat to the system.

```
1  public class Resource {
2
3     private int numServers; // Number of servers working in parallel
4     private int numBusy; // Number of servers that are busy
5     private Stat utilization; // Utilization of servers
6
7     public Resource(int size) {
8        numServers = size;
9        numBusy = 0;
10       utilization = new Stat();
11       DESimBuilder.rList.add(this);
12       DESimBuilder.sList.add(utilization);
13    }
```

The `isAvailable` method checks to see if an arriving Entity can use the Resource, and if so, the Entity will `seize` the Resource and tie up some of its capacity. When it is finished, it will `release` the Resource so that other Entities may use it. The Resource's `utilization` is automatically updated as Entities `seize` and `release` the Resource. Lastly, the `reset` method flushes the Resource and makes all of its capacity available again. It is used when restarting the system for additional replications.

```
1     // Check if new job can begin service immediately
2     public boolean isAvailable(double num) {
3        return (numServers-numBusy >= num);
4     }
5
6     // Seize (mark as busy) servers to begin working on new job
7     public boolean seize(double num) {
8        if (numServers-numBusy < num)
9           return false;
10       else {
11          utilization.recordCT(numBusy);
12          numBusy += num;
13          return true;
14       }
15    }
16
17    // Release (mark as available) servers after completed job
18    public boolean release(int num) {
19       if (numBusy < num)
20          return false;
21       else {
22          utilization.recordCT(numBusy);
23          numBusy -= num;
24          return true;
25       }
26    }
27
28    // Reset resource, release all servers
29    public void reset() {
30       numBusy = 0;
31    }
32
33 }
```

2.5. **DESimBuilder (context builder).** Now we take a look at `DESimBuilder`, our context builder that ties everything together for modeling the system. Already implemented is an $M/M/1$ queue, which illustrates how to incorporate each of the previously mentioned elements of DiscreteEventSim. In this section, we will focus only on the $M/M/1$ queueing system and save discussion of possible extensions for the next section.

We consider first our declarations, starting with `schedule`. This is our event calendar and will keep track of events that are scheduled to occur. Using the classic minimalist approach, events are only scheduled as they are needed and not all at once. For example, rather than scheduling the entire sequence of Entity arrivals to the system at the beginning, we only schedule the next arrival after the previous arrival event occurs.

Next, we declare our event types. In the implemented model, there are four different event types:

- `nextArrival`: a new Entity arrives and enters the system
- `nextDeparture`: an Entity finishes using the Resource and departs from the system
- `nextClear`: the warmup period for the current replication ends and all Stats are reset
- `nextEnd`: the current replication ends and the system is reset for the next replication

Note that we do not need to define an event for when an Entity is able to use the Resource. If the Resource is available when an Entity arrives, the Entity uses the Resource immediately. On the other hand, if the Resource is unavailable and there are Entities in the Queue, the Entity at the head of the Queue uses the Resource when the next departure occurs. This keeps with our minimalist approach to limit the number of events in the event calendar.

```
1  public class DESimBuilder implements ContextBuilder<Object> {
2
3      // Declare schedule and actions
4      static ISchedule schedule;
5      ISchedulableAction nextArrival, nextDeparture, nextClear, nextEnd;
```

To define the system, we declare all Queues and Resources along with any random number generators and additional Stats we may need. For the $M/M/1$ queueing system, we have just one Queue and one Resource, and one random number generator for each. The `arrivals` and `departures` variables will count the numbers of arrivals and departures, respectively, during each replication, and the five Stats will be used to tabulate the results across all replications.

The three lists - `qList`, `rList`, and `sList` - will be used to bundle all of the components of the modeled system so that they can be accessed together. While these lists are not essential for such a simple system, which has only one Queue, one Resource, and a handful of Stats, they can be invaluable for more complex systems with many components. It is important to note here that the Stats included in `sList` are only those associated with Queues and Resources, not the global Stats declared earlier in `DESimBuilder`. Because the Stats in `sList` are reset after each replication, it is important that the global Stats, which record the results of all replications, are not included in the list.

```
1     // Declare queues, resources, random number generators, and statistics
2     Queue mm1q;
3     Resource mm1r;
4     Exponential arrivalRNG, serviceRNG;
5     int arrivals, departures;
6     Stat arriveStat, departStat, lengthStat, waitStat, utilStat;
7
8     // Declare queue, resource, and stat lists
9     static List<Queue> qList;
10    static List<Resource> rList;
11    static List<Stat> sList;
```

The last thing to do before building the model is to declare the system and run parameters. In this model, `lambda` is the mean arrival rate of new Entities to the system, and `mu` is the mean service time of the Resource, or the average time that an Entity spends using the Resource. The run parameters include the number of `replications`, the `warmup` period during which the model runs but any Stats collected will not be recorded, and the `endTime`, or length of each replication. The variable `repCounter` keeps track of the current replication.

```
1     // Declare system parameters
2     double lambda; // Mean arrival rate
3     double mu; // Mean service time
4
5     // Declare run parameters
6     int replications; // Number of replications
7     int repCounter;
8     double warmup; // Warmup time for each replication
9     double endTime; // Length of each replication
```

The `build` method initializes all of the previously declared objects. The values for the system and run parameters are specified in the parameters.xml file in the DiscreteEventSim.rs folder. They can be modified either by editing the xml file directly or by navigating to the Parameters panel in the Eclipse display window and changing the values there.

```
1  @Override
2     public Context<Object> build(Context<Object> context) {
3
4        // Initialize everything
5        schedule = RunEnvironment.getInstance().getCurrentSchedule();
6        Parameters params = RunEnvironment.getInstance().getParameters();
7        replications = (Integer)params.getValue("replications");
8        warmup = (Double)params.getValue("warmup");
9        endTime = (Double)params.getValue("endTime");
10       lambda = (Double)params.getValue("lambda");
11       mu = (Double)params.getValue("mu");
12
13       arrivalRNG = RandomHelper.createExponential(lambda);
14       serviceRNG = RandomHelper.createExponential(mu);
15
16       qList = new ArrayList<Queue>();
17       rList = new ArrayList<Resource>();
18       sList = new ArrayList<Stat>();
19
20       mm1q = new Queue("FIFO");
21       mm1r = new Resource(1);
22
23       // Global statistics (not part of slist)
24       arriveStat = new Stat();
25       departStat = new Stat();
26       lengthStat = new Stat();
27       waitStat = new Stat();
28       utilStat = new Stat();
29
30       repCounter = 0; // Keep track of current replication
31
32       // Schedule first event
33       schedule.schedule(ScheduleParameters.createOneTime(0), this, "initialize");
34
35       return context;
36    }
```

Each event on our event calendar, `schedule`, corresponds to a method within `DESimBuilder`. When an event is pulled from the calendar, its corresponding method is called. The very first event, created when we initialized the model with the `build` method, calls the `initialize` method. This method resets the system by emptying all Queues, freeing all

Resources, and clearing all Stats. It also schedules the first arrival of an Entity to the system as well as the end of the warmup period ends and the end of the current replication.

```
1     // Initialize system before each replication
2     public void initialize() {
3        repCounter++;
4        arrivals = 0;
5        departures = 0;
6        for (Queue q:qList)
7           q.clear();
8        for (Resource r:rList)
9           r.reset();
10       for (Stat s:sList)
11          s.initialize();
12
13       double firstArrival = schedule.getTickCount()+arrivalRNG.nextDouble();
14       nextArrival = schedule.schedule(ScheduleParameters.createOneTime(
15         firstArrival, 1), this, "arrive");
16       nextClear = schedule.schedule(ScheduleParameters.createOneTime(
17         schedule.getTickCount()+warmup, ScheduleParameters.LAST_PRIORITY),
18           this, "clearStats");
19       nextEnd = schedule.schedule(ScheduleParameters.createOneTime(
20         schedule.getTickCount()+endTime, ScheduleParameters.LAST_PRIORITY),
21           this, "end");
22    }
```

When an Entity `arrives`, it checks to see if the Resource is available. If it is, then the Entity begins using the Resource right away; otherwise, it enters the Queue. The next Entity arrival is also scheduled.

When an Entity finishes using the Resource, it `departs` from the system. If there are any Entities waiting in the Queue, the one at the head leaves the Queue and begins using the Resource.

```
1     // Arrival event
2     public void arrive() {
3        arrivals++;
4        Entity e = new Entity();
5        if (mm1r.isAvailable(1)) {
6           mm1q.skip();
7           mm1r.seize(1);
8           double departTime = schedule.getTickCount()+serviceRNG.nextDouble();
9           if (departTime < nextEnd.getNextTime()) /* This conditional (and others
10            like it) can be removed once removeAction is implemented */
11             nextDeparture = schedule.schedule(ScheduleParameters.createOneTime(
12               departTime, 1), this, "depart", e);
13       }
14       else
15          mm1q.enqueue(e);
16       double arriveTime = schedule.getTickCount()+arrivalRNG.nextDouble();
17       if (arriveTime < nextEnd.getNextTime())
18          nextArrival = schedule.schedule(ScheduleParameters.createOneTime(
19            arriveTime, 1), this, "arrive");
20    }
21
22    // End of service event
23    public void depart(Entity e) {
24       Entity next = mm1q.pop();
25       if (next != null) {
26          double departTime = schedule.getTickCount()+serviceRNG.nextDouble();
27          if (departTime < nextEnd.getNextTime())
28             nextDeparture = schedule.schedule(ScheduleParameters.createOneTime(
29               departTime, 1), this, "depart", next);
30       }
31       else
32          mm1r.release(1);
33       departures++;
34    }
```

The `clearStats` method is called at the end of each warmup period and resets all Stats for the current replication. Warmup periods are commonly used when performing steady-state analysis of a system since the system state is often much different at the beginning when it is initialized than after it has been allowed to run for some time. At the end of each replication, the `end` method is called and records the results of the replication to the

global Stats. If it is the final replication, a summary of the global Stats is printed to the console.

```
1     // Clear statistics after warmup period
2     public void clearStats() {
3        for (Stat s:sList)
4           s.initialize();
5        arrivals = 0;
6        departures = 0;
7     }
8
9     // Record and print statistics after each replication; reset system
10    //   for next replication
11    public void end() {
12       /*
13        * To be added when removeAction bug is fixed
14       if (nextarrival.getNextTime() > schedule.getTickCount())
15          schedule.removeAction(nextarrival);
16       if (nextdeparture.getNextTime() > schedule.getTickCount())
17          schedule.removeAction(nextdeparture);
18       */
19       recordGlobalStats();
20       printStats();
21       if (repCounter < replications)
22          initialize();
23       else {
24          System.out.println("Mean arrival rate:  "+
25            arriveStat.getAverage()/(endTime-warmup));
26          System.out.println("Mean service rate:  "+
27            departStat.getAverage()/(endTime-warmup));
28          System.out.println("Mean queue length:  "+
29            lengthStat.getAverage());
30          System.out.println("Mean waiting time in queue:  "+
31            waitStat.getAverage());
32          System.out.println("Mean resource utilization:  "+
33            utilStat.getAverage());
34       }
35    }
```

The final two methods - `recordGlobalStats` and `printStats` - are called from within the `end` method and are used for handling the five global Stats.

```
1     // Save statistics from current replication
2     public void recordGlobalStats() {
3        arriveStat.recordDT(arrivals);
4        departStat.recordDT(departures);
5        lengthStat.recordDT(sList.get(0).getAverage());
6        waitStat.recordDT(sList.get(1).getAverage());
7        utilStat.recordDT(sList.get(2).getAverage());
8     }
9
10    // Print statistics to console as comma-separated list
11    public void printStats() {
12       System.out.print(arrivals+","+departures);
13       for (Stat s:sList)
14          System.out.print(","+s.getAverage());
15       System.out.println();
16    }
17
18 }
```

3. Beyond the M/M/1 Queue